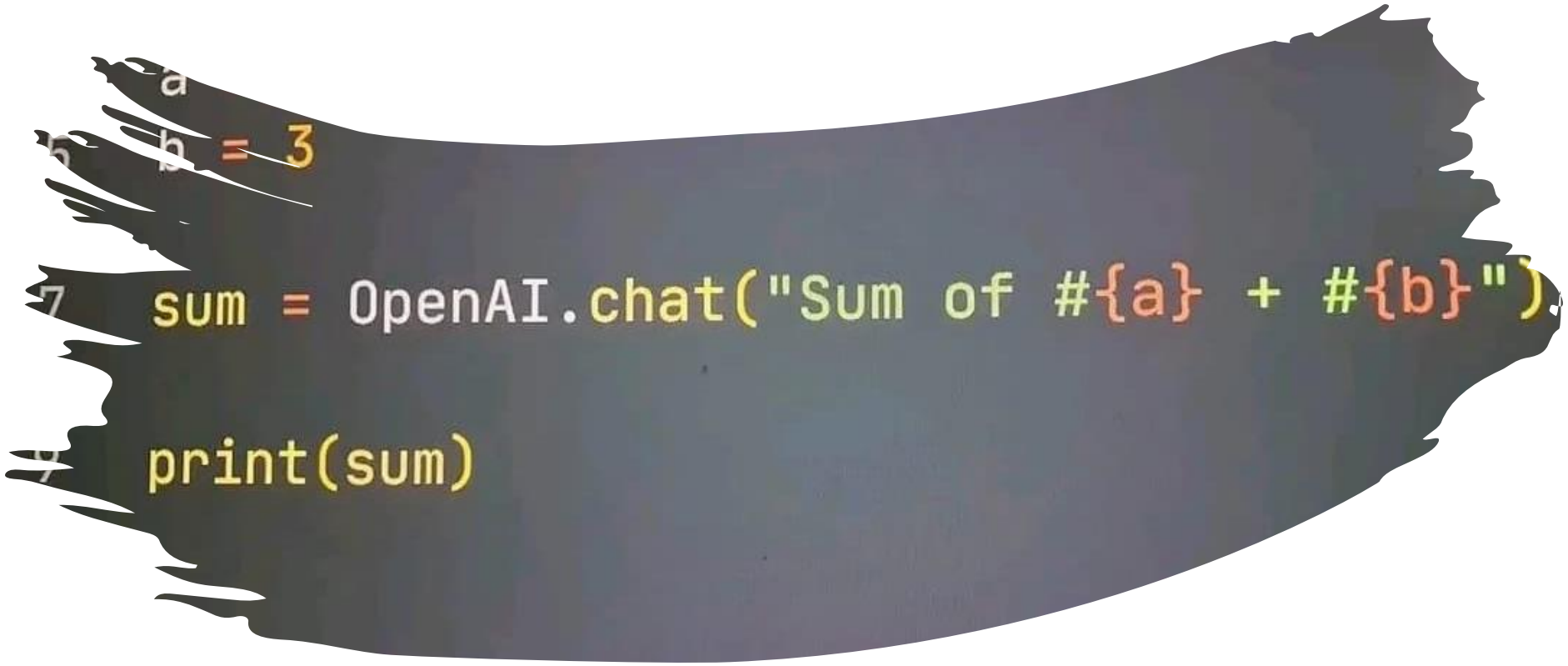
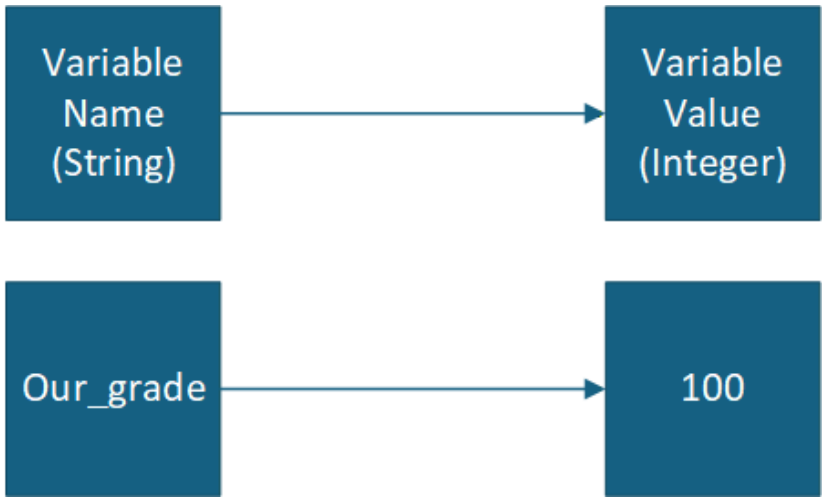



# Mutation and State: From Substitution to Store

By Emily Herman, Megan Ruiz, Erik Ellams, Nickolas Diaz

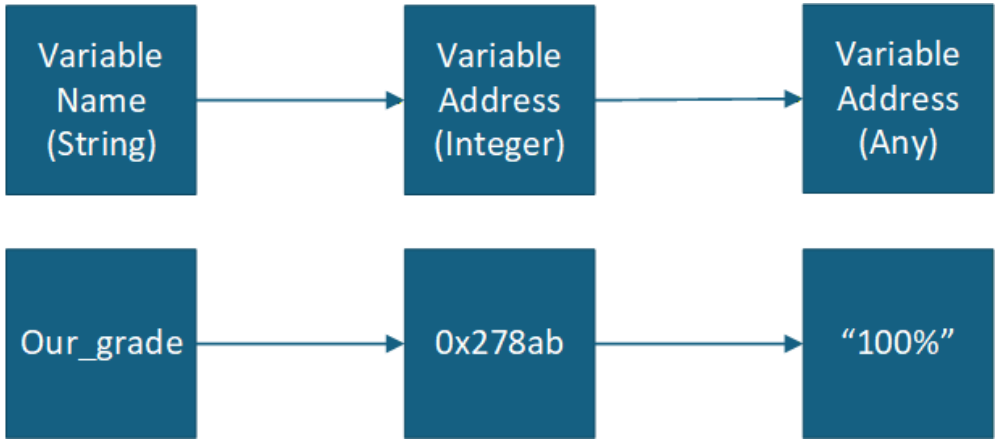
# Introduction

- Previously we have worked with the substitution model: where each variable name is bounded with its value using a dictionary.
- This approach has some problems:
- It is not very flexible; it cannot handle multiple data types (Integer/Float/Boolean).
- It is immutable: a variable's value cannot be changed
- It does not represent how actual programing languages store their variables like Python, Java, or C.

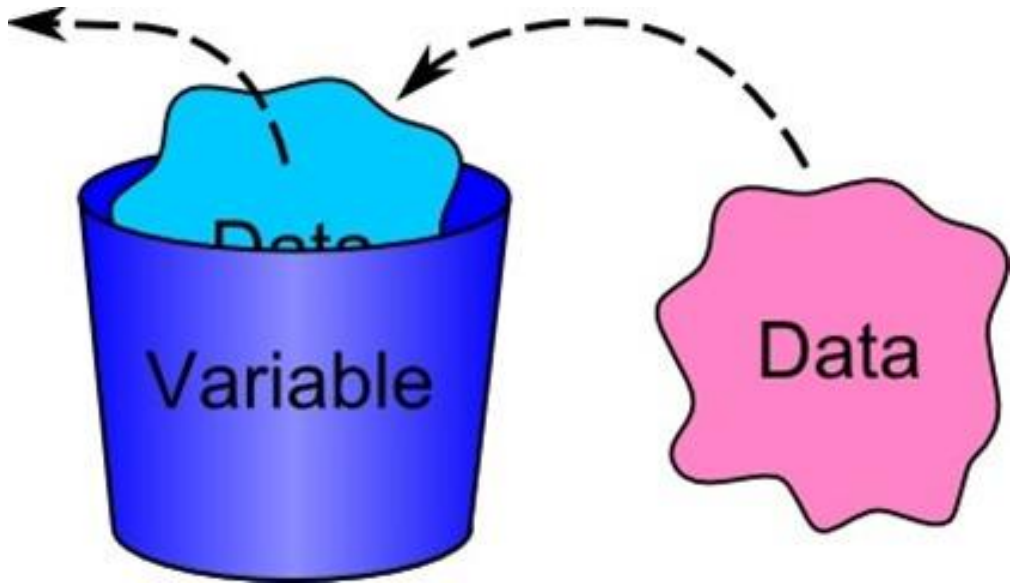


# The environment and store model

- This is a better approach where we use a dictionary to store the variable's name as the key and the address of the variable.
- The variable's true location is in memory, in this project the memory will be a vector like
  {define MEMORY (make-vector 100 -1)}
- This solves the problems of flexibility and immutable nature, changing the variable means changing the memory instead of the dictionary. In addition, variables can be of different sizes.



# Set Functionality




- The set! functionality evaluates the expression, finds the location of the variable x in the environment, and updates the store so that location now holds the new value.
- Example Usage: {set! variable_name experesion}
- {test (run `{let1 {x 3}{set! {x 3}}}) (numV 3)}
- You first have to use let1 before using set! Or you will get an "unbound variable" error.

# Set implementation helper functions

```
; defines memory a vector of 100
{define MEMORY (make-vector 100 -1)}
; incrementing pointer for the memory, example 0, 1, 2, 3 ...
{define next-addr 0}

; allocates a new location
; returns the address
{define (create-num [value : Number]) : Number
  (let ([cur-addr next-addr])
    (begin
    (vector-set! MEMORY cur-addr value)
    (set! next-addr (add1 next-addr))
    cur-addr))}

; takes in the address and returns the address's value
{define (read-num [address : Number]) : Number
  (vector-ref MEMORY address)}

; takes in an address and value and sets it
{define (set-num [value : Number] [address : Number]) : Void
  (vector-set! MEMORY address value)}
```

```
{define-type Value
  [numV (n : Number)]
  [funV (var : Symbol) (body : Exp) (nv : Env)]}
```

```
{define-type Binding
  [bind (name : Symbol) (val : Value)]}

{define-type-alias Env (Listof Binding)}
{define empty-env : Env empty}
```

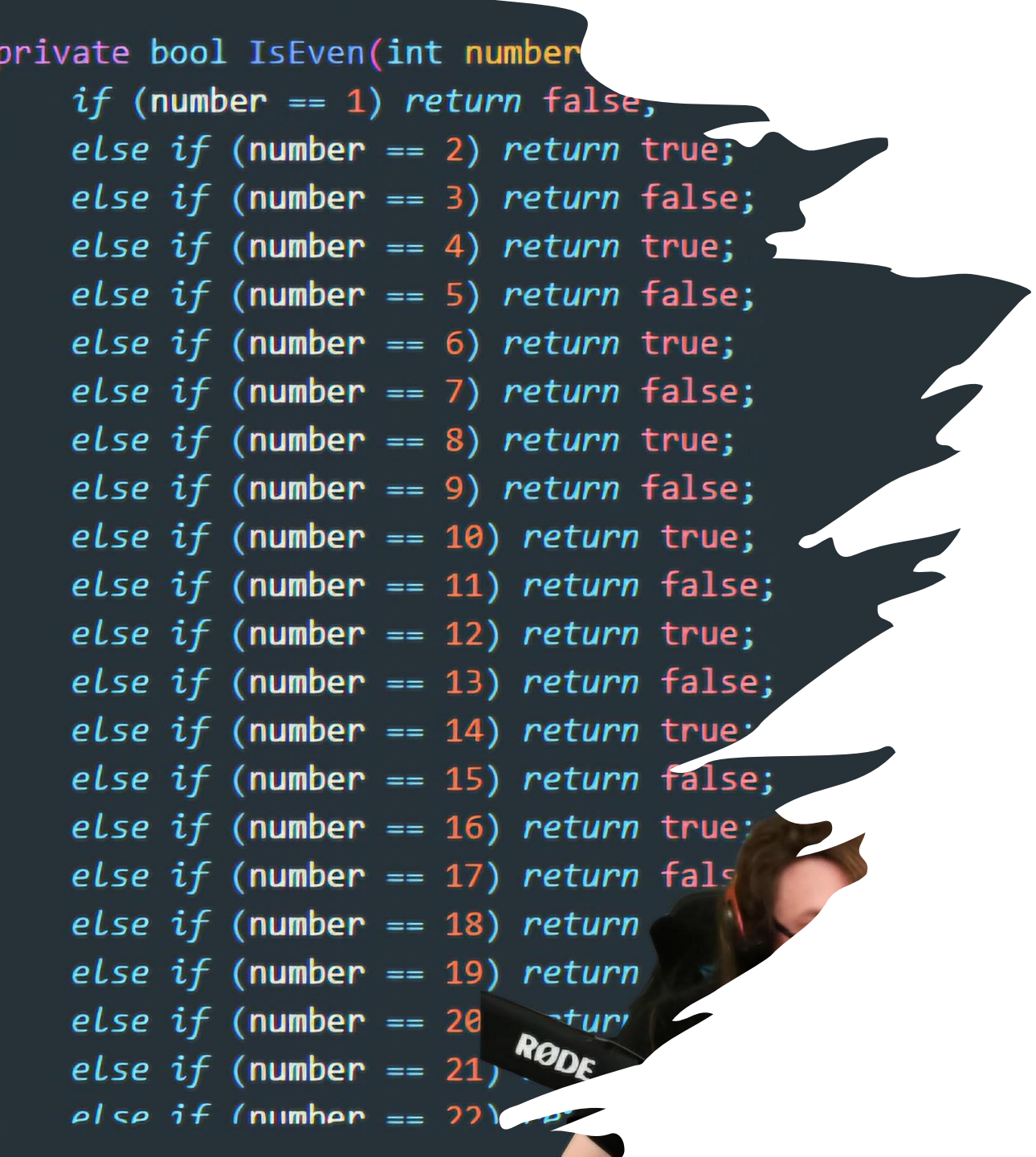
# Part 2

Modifying existing functions

To implement it modification to the interpreter, desugarer, parser and expression functions is needed.

Also, the storing of the lambda function is out of the project's scope.

```
{define (lookup [name : Symbol] [env : Env]) : Value
  (cond
    [(empty? env)
     (error name "unbound variable")]
    [(symbol=? name (bind-name (first env)))
     (type-case Value (bind-val (first env))
       [(numV n) (numV (read-num n))]
       [(funV var body nv) (funV var body nv)])]
    [else
     (lookup name (rest env))])} ; empty env

{define (extend [env : Env] [name : Symbol] [val : Value]) : Env
  (type-case Value val
    [(numV n)
     (let ([new-address (create-num n)]) ; cheks if val is a number then calls create-num returing it's address
       (cons (bind name (numV new-address)) env))]; address is stored on the env instead of the number value
    [(funV _var _body _nv) (cons (bind name val) env)])}

; checks if
{define (set-binding [env : Env] [name : Symbol] [new-val : Number]) : Value
  (cond
    [(empty? env)
     (error name "unbound variable")]
    [(symbol=? name (bind-name (first env)))
    ; env will return a Value even though it is guaranteed to be a numV
    ; plait forces a type-case
     (type-case Value (bind-val (first env))
       [(numV addr)
        (begin
          (set-num new-val addr)
          (numV new-val))]
       [(funV _var _body _nv)
        (error 'set1E "cannot assign to a function")])]
    [else
     (set-binding (rest env) name new-val)])} ; empty env
```


private bool IsEven(int number
if (number == 1) return false;
else if (number == 2) return true;
else if (number == 3) return false;
else if (number == 4) return true;
else if (number == 5) return false;
else if (number == 6) return true;
else if (number == 7) return false;
else if (number == 8) return true;
else if (number == 9) return false;
else if (number == 10) return true;
else if (number == 11) return false;
else if (number == 12) return true;
else if (number == 13) return false;
else if (number == 14) return true;
else if (number == 15) return false;
else if (number == 16) return true;
else if (number == 17) return fals
else if (number == 18) return
else if (number == 19) return
else if (number == 20
else if (number == 21)
RØDE

# Begin Functionality

- The begin evaluates a list of expressions from left to right and returns the value of the last one. The intermediate values are discarded, but their side effects (such as assignments) persist in the store.
- Example Usage: {begin expr1 expr2 … exprN}
- The expression the is return is exprN.
- Evaluates one or many expressions

# Begin Implementation

```
; AST
{define-type Exp
  [beginE (es    : (Listof Exp))]
```

```
{define (interp [e : Exp] [nv : Env]) : Value
  (type-case Exp e
    [(beginE es)
     (if (empty? es)
         (error 'interp "begin: empty sequence")
         (eval-begin es nv))]
```

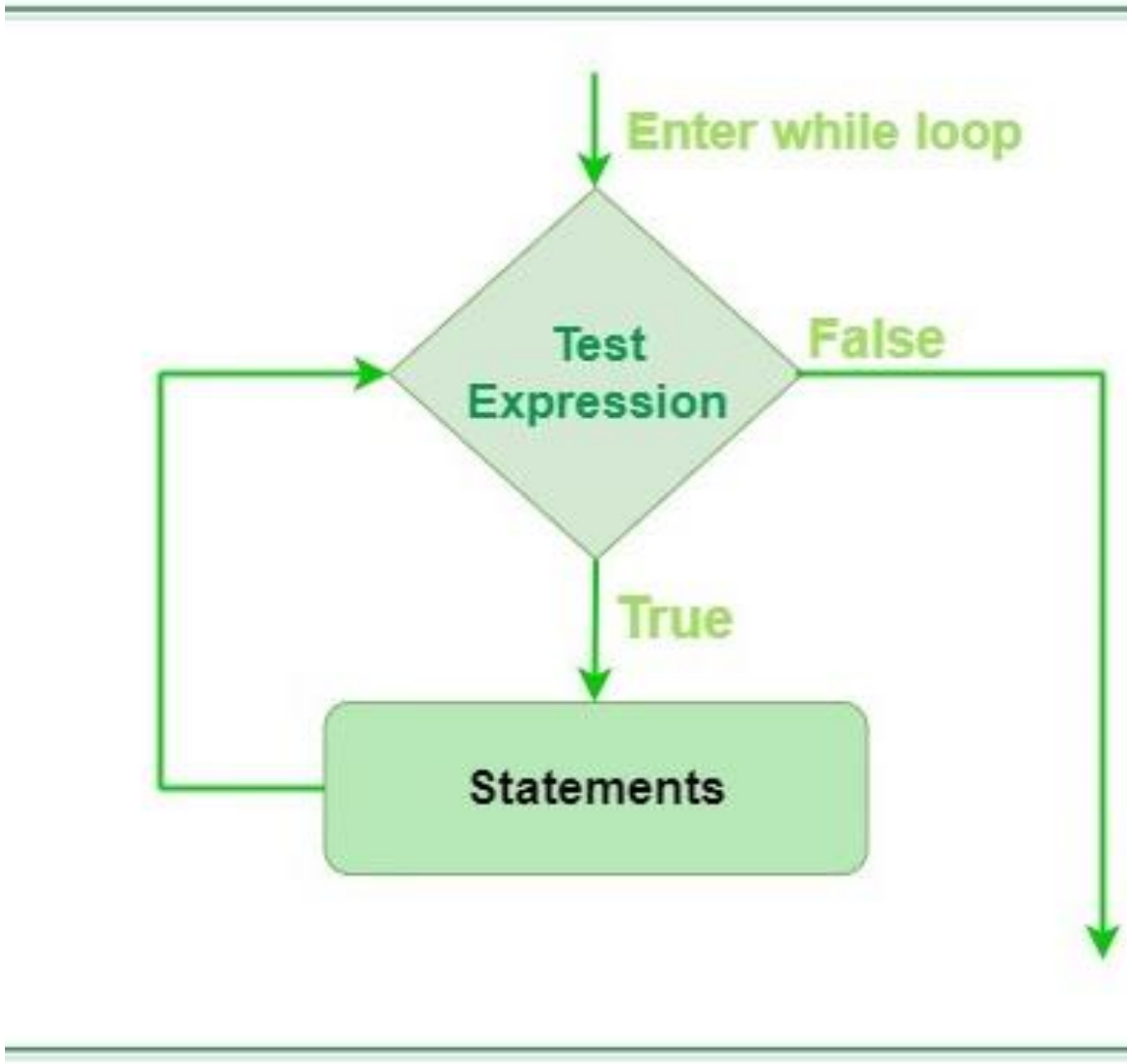
```
{define (parse [s : S-Exp]) : Exp
  [(and (s-exp-symbol? (first l))
        (symbol=? 'begin (s-exp->symbol (first l))))
   (beginE (map parse (rest l)))]
```

```
{define (desugar [e : Exp]) : Exp
  (type-case Exp e
    [(beginE es)     (beginE (map desugar es))]
```

```
{define (eval-begin [es : (Listof Exp)] [nv : Env]) : Value
  (cond
    [(empty? (rest es)) (interp (first es) nv)]
    [else
     (begin
       (interp (first es) nv)
       (eval-begin (rest es) nv))])}
```

# While Functionality

- The while loop takes in a condition and a body,
- where if the condition is true the body is running else it goes to the next condition.
- When the body is finished, it goes back to the condition, forming a loop.
- Example Usage: {while condition body}

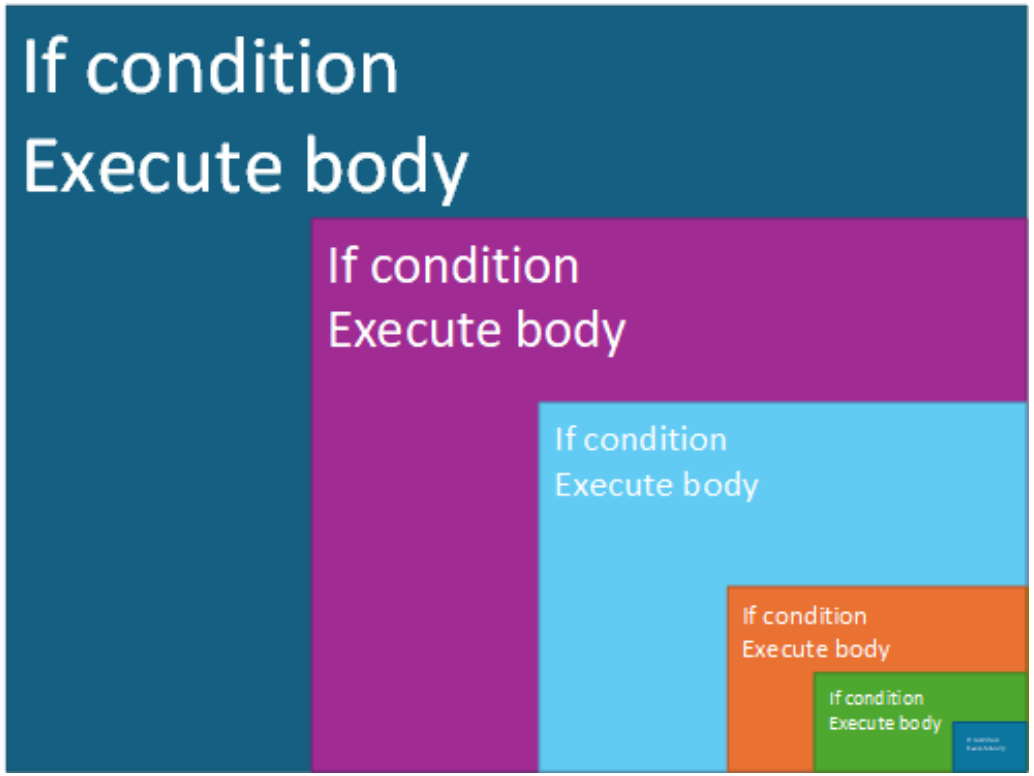


# While Functionality Part 2

- It can be implemented solely through the desugurer
- {while cond body} can become
- {if cond {

 begin body {while cond body}

 }
- Notice that the while loop desugars into a recursive structure: if the condition is true, execute the body, then repeat; otherwise, return 0.



# While Implementation Part 3

- The assignment said to implement the while loop in the desugarer, however we tried and failed to do so.
- There is no way to implement the while loop solely in the desugarer as there is no way to check if a conditional. The code goes on to infinity.

```
[(whileE c b)
      (begin
        (display c)
        (ifE (desugar c)
             (beginE (list (desugar b)
(desugar (whileE c b))))
             (numE 0)))]
          [(andE l r)
          (desugar (ifE l r (numE 0)))]
```

# While Implementation Part 4

- We ended up implementing the while loop straight in the interpreter
- Similar to the one above but actually having a base case

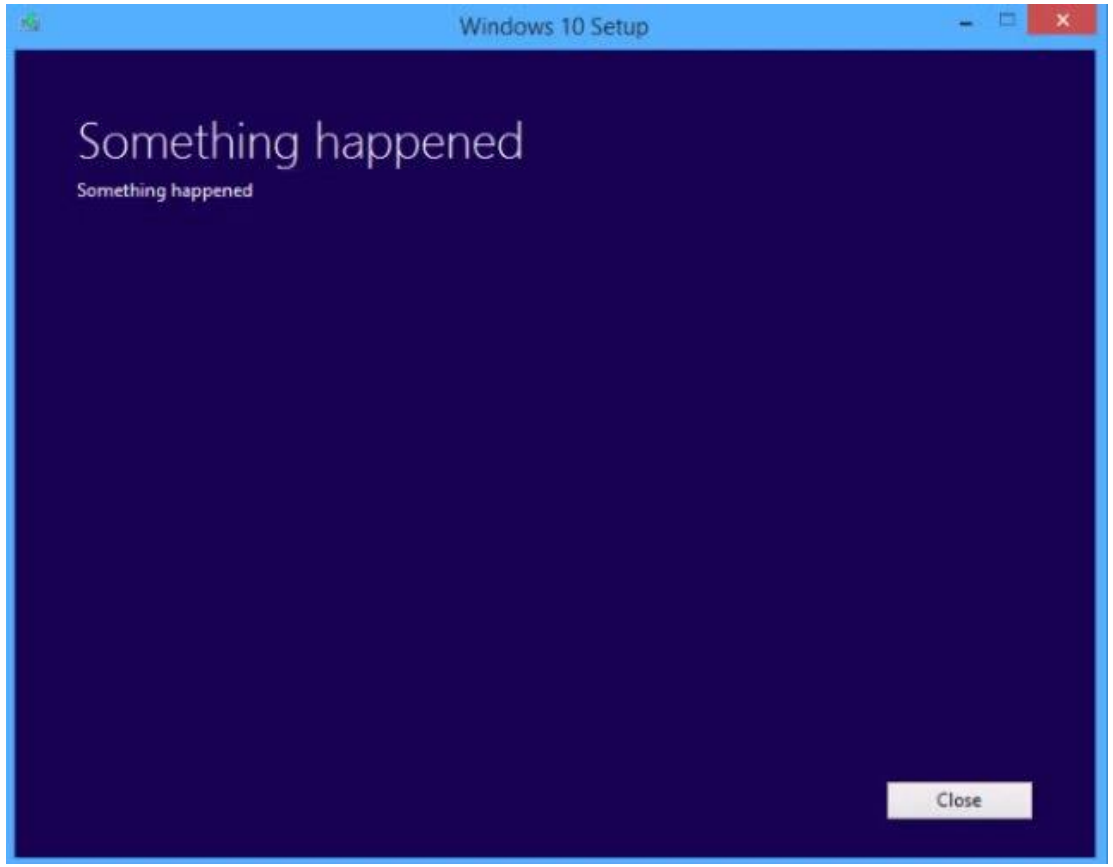
```
[(whileE c b)
 (if (equal? (interp c nv) (numV 0))
     (numV 0)
     (begin
       (interp b nv)
       (interp (whileE c b) nv)))]
```

# Test Cases

```
{test/exn (run `{set! x {5}}) "unbound"}

{test (run `{let1 {x 3} {set! x {3}}})  (numV 3)}

{test (run `{let1 {x 3} {set! x {{lam x {+ x x}} 3}}})  (numV 6)}
```

# Test Cases Part 2

```
{test (run `{let1 {x 5}
                  {begin {set! x {10}} x}})
      (numV 10)}

{test (run `{let1 {x 1}
                  {begin {set! x {+ x 1}}
                         {set! x {+ x 1}}
                         x}})
      (numV 3)}
```



# Test Cases Part 3

```
{test (run `{let1 {x 0}
                  {let1 {inc {lam _ {set! x {+ x 1}}}}
                        {begin {inc 0} {inc 0} {inc 0} x}}}
           ) (numV 3)}
```

```
{test (run `{let1 {x 0}
                  {let1 {i 1}
                        {begin
                          {while
                           {if {+ i {- 0 6}} 1 0}
                           {begin {set! x {+ x i}}
                                  {set! i {+ i 1}}}} x}}})
      (numV 15)}
```